Performance Analysis

Professor Yeali S. Sun
Department of Information Management
National Taiwan University
Performance Analysis

To obtain estimates of time and space of a program execution, independent of machine.

Performance measurements
- To obtain machine dependent execution times

Space complexity
- The amount of memory needed by a program to run to completion

Time complexity
- The amount of computing time needed by a program to run to completion
Space Complexity

**Fixed space requirement**
- The required memory that is independent of the number and size of the program’s inputs and outputs
- E.g., instruction (code) space, simple variables, fixed-size structured variables and constants

```c
main {
    int i, j, k;
    float x, y, z;
    struct {
        char name[Max_nameSize];
        ...
    } student_record;
    struct student_record *stu_record_ptr;

    ...
    stu_record_ptr = alloc(struct record, size of struct)
    call func_selectCourse(department_id);
    ...
}
```
Space Complexity (cont’d)

**Variable space requirements**
- memory that is dependent upon the number and size of the program’s inputs and outputs.
- e.g., space needed by structured variables for a particular instance, \( I \), of the problem being solved.
- it depends on some characteristics of the instance \( I \).
- e.g., the number, size, and values of the inputs and outputs of \( I \).

**Notations**
\[ S(P) = c + S_P(I) \]
Space Complexity: discussion

- More complex than time complexity
- It strongly depends on data structures design
  - It affects the computing time complexity
- In general-purpose computing platform
  - Memory, cache, secondary storage
- In embedded system
  - Limited memory (SDRAM, FLASH (ROM))
  - No secondary storage
How to compute the time complexity of a program?

\[ T(P) \]

- the total amount of time, taken by a program, \( P \), which is the sum of compile time and its run (execution) time.

Compile time
- is independent of the instance characteristics
- no recompilation is needed after the program is verified to run correctly.

Execution time, \( T_p \)
- is dependent upon the attributes of compilers
- exact counting of instruction times, e.g., (not practical)

\[ T_p(n) = c_a \text{ADD}(n) + c_s \text{SUB}(n) + c_i \text{LDA}(n) + c_{s_i} \text{STA}(n) \]
Time Complexity (cont’d)

Counting of the number of operations that a program performs, i.e., “step” count

- A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics
- difficult and inexact, e.g.,

\[
x = y \quad \text{vs.} \quad x = y + z + (x/y) + a*b/0.8
\]

All count as one step!
Asymptotic Notation

Motivation
- To compute the time complexity of a program
- “step” count:
  * difficult and inexact
  * not very useful for comparison purpose
- “bounds” – useful and adequate in many situations
  i.e.

\[ c_1 n \leq T_p(n) \leq c_2 n^2 \]
Asymptotic Notation

Example:

\[ P_1 : T_{p_1} (n) = c_2 n^2 + c_1 n \quad \text{(Exact)} \]
\[ P_2 : T_{p_2} (n) = c_3 n \]

for \( n > 1 \) \( P_2 \) is faster than \( P_1 \)
Figure 9-3
A comparison of growth-rate functions: (a) in tabular form;
(b) in graphical form
**Big “Oh”**

**Definition:**

\[ f(n) = O(g(n)) \iff \exists c > 0, n_0 > 0 \]

such that \( \forall n, n \geq n_0, f(n) \leq cg(n) \)

---

**Examples**

\[ \begin{align*}
3n + 2 & \leq 4n \\
f(n) & = O(g(n)) \\
3n + 3 & = O(n), c = 4, n \geq 2 \\
100n + 6 & = O(n), c = 101, n \geq 6 \\
10n^2 + 4n + 2 & = O(n^2), c = 11, n \geq 5 \\
6 \times 2^n + n^2 & = O(2^n), c = 7, n \geq 4
\end{align*} \]
Big “Oh”

**Relation**

\[ O(2) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n) \]

**An upper bound**

To be informative, \( g(n) \) should be as small a function of \( n \) as possible for which the statement is true.

**Theorem:**

If \( f(n) = a_m n^m + \cdots + a_1 n + a_0 \)

then \( f(n) = O(n^m) \)

\[ f(n) \leq \sum_{i=0}^{m} |a_i| n^i \]

**proof:**

\[ \leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \]

\[ \leq n^m \sum_{i=0}^{m} |a_i| \quad \forall n \geq 1 \]
**Omega** \( \Omega \)

**Definition:**

\[ f(n) = \Omega(g(n)) \iff \exists c > 0, n_0 > 0 \]

such that \( \forall n, n \geq n_0, f(n) \geq cg(n) \)

A lower bound to be informative, \( g(n) \) should be as large a function of \( n \) as possible for which the statement is true.

**Examples**

\[ f(n) = \Omega(g(n)) \quad c \quad n \]

\[ 3n + 2 = \Omega(n), \quad c = 3, \quad n \geq 1 \]

\[ 10n^2 + 4n + 2 = \Omega(n^2), \quad c = 10, \quad n \geq 1 \]
**Theta \( \Theta \)**

- **Exact**
- **Definition:**
  
  \[
  f(n) = \Theta(g(n)) \quad \text{iff} \quad \exists c_1, c_2, n_0 > 0
  \]

  such that \( \forall n, n \geq n_0, c_1g(n) \leq f(n) \leq c_2g(n) \)

- **Examples**

  \[
  \begin{align*}
  f(n) &= \Theta(g(n)) \quad c_1 = 1, \quad c_2 = 2, \quad n_0 \\
  3n + 2 &= \Theta(n) \quad c_1 = 3, \quad c_2 = 4, \quad n_0 \geq 2 \\
  6 \times 2^n + n^2 &= \Theta(2^n) \quad c_1 = 6, \quad c_2 = 7, \quad n_0 \geq 4
  \end{align*}
  \]
When \( N \) is large

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \log_2 N )</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>( N )</td>
<td>10</td>
<td>(10^2)</td>
<td>(10^3)</td>
<td>(10^4)</td>
<td>(10^5)</td>
<td>(10^6)</td>
</tr>
<tr>
<td>( N \times \log_2 N )</td>
<td>30</td>
<td>664</td>
<td>9,965</td>
<td>(10^5)</td>
<td>(10^6)</td>
<td>(10^7)</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>(10^2)</td>
<td>(10^4)</td>
<td>(10^6)</td>
<td>(10^8)</td>
<td>(10^{10})</td>
<td>(10^{12})</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>(10^3)</td>
<td>(10^6)</td>
<td>(10^9)</td>
<td>(10^{12})</td>
<td>(10^{15})</td>
<td>(10^{18})</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>(10^3)</td>
<td>(10^{30})</td>
<td>(10^{301})</td>
<td>(10^{3,010})</td>
<td>(10^{30,103})</td>
<td>(10^{301,030})</td>
</tr>
</tbody>
</table>
Asymptotic Notations: Properties

**Theorem 1.3**

If \( f(n) = a_m n^m + \cdots + a_1 n + a_0 \) and \( a_m > 0 \), then \( f(n) = \Omega(n^m) \)

**Theorem 1.4**

If \( f(n) = a_m n^m + \cdots + a_1 n + a_0 \) and \( a_m > 0 \), then \( f(n) = \Theta(n^m) \)
Performance Analysis

Example#1: Matrix Addition

\[ \begin{align*}
    c &= a + b \quad M \times N \\
    c_{ij} &= a_{ij} + b_{ij}
\end{align*} \]

<table>
<thead>
<tr>
<th>Statement</th>
<th>Asymptotic complexity</th>
</tr>
</thead>
</table>
| void add(int a[][MAX_SIZE]...)  
  
  ```
  int i, j;
  for (i=0; i<rows; i++)
    for (j=0; j<cols; j++)
      c[i][j] = a[i][j] + b[i][j];
  ``` |
| 1  
  1  
  \( \Theta(\text{rows}) \)  
  \( \Theta(\text{rows} \times \text{cols}) \)  
  \( \Theta(\text{rows} \times \text{cols}) \) |

Total  
\( \Theta(\text{rows} \times \text{cols}) \)
Example#2: Magic square program

```c
#include <stdio.h>
#define MAX_SIZE 15  /*maximum size of square */

void main(void)
/*construct a magic square, iteratively*/
{
    static int square[MAX_SIZE][MAX_SIZE];
    int i, j, row, column;           /*indices*/
    int count;                        /*counter*/
    int size;                         /*Square size*/

    printf("Enter the size of the square:");
    scanf("%d", &size);
    /*check for input errors */
    if (size<1 || size>MAX_SIZE+1) {
        fprintf(stderr, "Error! Size is out of range\n");
        exit(1);
    }
    cont'd
```
if(!(size%2)) {
    fprintf(stderr, "Error! Size is even\n");
    exit(1);
}

for (i=0;i<size;i++)
    for(j=0;j<size;j++)
        square[i][j]=0;

square[0][(size-1)/2]=1;           /*middle of first row*/

i=0;                     /* i and j are current position*/
j=(size-1)/2;

for(count=2;count<=size*size;count++){
    row=(i-1<0)?(size-1):(i-1); /*up*/
    column=(j-1<0)?(size-1):(j-1);    /*left*/
    if(square[row][column]) /*down*/
        i=(++i) % size;
    else{ /*square is unoccupied*/
        i=row;
        j=(j-1<0)?(size-1):--j;
    }
    square[i][j]=count;
}

/* output the magic square*/
printf("Magic Square of size %d: \n\n", size);
for(i=0;i<size;i++){
    for(j=0;j<size;j++)
        printf("%5d", square[i][j]);
    printf("\n");
}

printf("\n\n");
Example#3: Searching an ordered list

```c
int binsearch(int list[], int searchnum, int left, int right) {
    /* search list[0] <= list[1] <= ..... <= list[n-1] for searchnum. Return its position if found. Otherwise return -1 */
    int middle;
    while (left <= right) {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: left = middle + 1; break;
            case 0: return middle;
            case 1: right = middle - 1;
        }
    }
    return -1;
}
```

Asymptotic Complexity

```
\[ \log_{10} a = \log_2 a / \log_2 10 = c_1 \log_2 a \]
```

\[ n \quad 1 \]
\[ n/2 \quad 1 \]
\[ n/4 \quad 1 \]
\[ n/8 \quad 1 \]
\[ \ldots \]
\[ 1 \quad 1 \]
\[ \log_2 n + 1 \]

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Example#4: Recursive permutation

```c
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n] */
{
    int j, temp;
    if (i == n) {
        for (j = 0; j <= n; j++)
            printf ( "%c", list[j] );
        printf ( " " );
    } else {
        /* list[i] to list[n] has more than one permutation,
            generate these recursively */
        for (j = i; j <= n; j++) {
            SWAP(list[i], list[j], temp);
            perm( list, i+1, n );
            SWAP(list[i], list[j], temp);
        }
    }
}
```
Permutation

\[
T_{\text{perm}}(i, n) = \Theta(\Theta(n - i + 1) \cdot T_{\text{perm}}(i + 1, n))
\]

\[
T_{\text{perm}}(i + 1, n) = \Theta(\Theta(n - (i + 1) + 1) \cdot T_{\text{perm}}(i + 2, n))
\]

\[
= \Theta(\Theta(n - i) \cdot T_{\text{perm}}(i + 2, n))
\]

\[
\vdots
\]

\[
= \Theta(n - n + 2 + 1)
\]

\[
T_{\text{perm}}(n - 2, n) = \Theta(\Theta(3) \cdot T_{\text{perm}}(n - 1, n))
\]

\[
T_{\text{perm}}(n - 1, n) = \Theta(\Theta(2) \cdot T_{\text{perm}}(n - 1, n))
\]

\[
T_{\text{perm}}(n, n) = \Theta(n)
\]
Hanoi Tower Problem
Tower of Hanoi Problem

```cpp
void hanoiTower(int count, char source, char destination, char spare)
{
    if (count == 1)
        cout << "Move top disk from pole " << source << " to pole " << destination << endl;
    else {
        hanoiTower(count - 1, source, spare, destination);
        hanoiTower(1, source, destination, spare);
        hanoiTower(count - 1, spare, destination, source);
    }
}
```
Tower of Hanoi

Complexity

\[ T_{\text{hanoi}}(n) = \Theta (T_{\text{hanoi}}(n-1) + T_{\text{hanoi}}(n-1)) \]

\[ T_{\text{hanoi}}(n-1) = \Theta (T_{\text{hanoi}}(n-2) + T_{\text{hanoi}}(n-2)) \]

...\[ T_{\text{hanoi}}(2) = \Theta (T_{\text{hanoi}}(1) + T_{\text{hanoi}}(1)) \]
\[ T_{\text{hanoi}}(1) = \Theta (1) \]

\[ \Theta (2^n) \]

// void hanoiTower (int count, char source, char destination, char spare)
// {
//     if (count==1)
//     {
//         cout << "Move top disk from pole " << source << " to pole " << destination << end1;
//     }
//     else {
//         hanoiTower (count-1, source, spare, destination);
//         hanoiTower (1, source, destination, spare);
//         hanoiTower (count-1, spare, destination, source);
//     }
// }
Performance Measurement

- To measure the computing time and memory usage of a program by running it on a computer.

  - Compared to “performance analysis” in which a program’s time and space complexity are assessed (i.e. estimated)

In C or C++

- Method 1: CLOCK
  * use internal processor time
  * divide it by CLOCK_PER_SEC (a built-in constant tick/sec)

cont’d
Performance Measurement

- #include <time.h>

* the exact syntax of the timing function depends on the operating systems and the computers in use
Introduction
#include <iostream.h>
#include <time.h>

void main()
{

double sum=1.0;

clock_t start, finish;

double duration;

start=clock();
test();  // 在這裡可以放置你要測量的函式
finish=clock();

duration=double(finish - start)/CLOCKS_PER_SEC;

cout<<"Total time is  "<<duration<<"/n";
}

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
More about “Clock” ...

- Clock returns the number of clock ticks of elapsed processor time.

- The returned value is the product of the amount of time that has elapsed since the start of a process and the value of the CLOCK_PER_SEC constant.
More about “Clock” ... (cont’d)

- The data type “clock_t” is defined in “time.h”

- In Intel processors, 
  \texttt{CLOCK\_PER\_SEC}=18.2, i.e., the precision is 1/18.2 second.
The end. 😊
Homework 2008 (1/2)

1. Show that the following statements are correct.

(a) \(5n^2 - 6n = \Theta(n^2)\)

(b) \(n! = O(n^n)\)

(c) \(2n^2 - n \log n = \Theta(n^2)\)

(d) \(6n^3 / (\log n + 1) = \Theta(n^{1.001})\)

(e) \(10n^3 + 15n^4 + 100n^2 2^n = O(n^2 2^n)\)
Homework 2008 (2/2)

2. Write an iterative version of the “Hanoi Tower” Problem. (Hint: simulate the recursive execution of the operating system)